Science of Computer Programming 77 (2012) 533–550

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

JCML: A specification language for the runtime verification of Java Card programs

Umberto Souza da Costa[a], Anamaria Martins Moreira[a], Martin A. Musicante[a], Plácido A. Souza Neto[a,b,*]

[a] DIMAp - Universidade Federal do Rio Grande do Norte, Campus Universitário, Lagoa Nova, Natal, RN, Brazil

[b] DIETInf - Instituto Federal do Rio Grande do Norte, Caixa Postal 1559, 59.015-000, Natal, RN, Brazil

ARTICLE INFO

Article history:
Received 17 February 2009
Received in revised form 16 March 2010
Accepted 19 March 2010
Available online 7 April 2010

Keywords:
JML
Java Card
JCML
Runtime verification

ABSTRACT

Java Card is a version of Java developed to run on devices with severe storage and processing restrictions. The applets that run on these devices are frequently intended for use in critical, highly distributed, mobile conditions. They are required to be portable and safe. Often, the requirements of the application impose the use of dynamic, on-card verifications, but most of the research developed to improve the safety of Java Card applets concentrates on static verification methods. This work presents a runtime verification approach based on Design by Contract to improve the safety of Java Card applications. To this end, we propose JCML (Java Card Modelling Language) a specification language derived from JML (Java Modelling Language) and its implementation: a compiler that generates runtime verification code. We also present some experiments and quality indicators. This paper extends previous published work from the authors with a more complete and precise definition of the JCML language and new experiments and results.

1. Introduction

The Java Card programming language [32] is a version of Java. Its programs are intended to run on very restricted architectures such as Smart Cards, SIM cards or security tokens. For this reason, many features and constructors of Java are not present in Java Card. These include some primitive types such as `float` and most library classes. A specific version of the Java Virtual Machine (JVM) has been defined to run Java Card applets [9]. These applets are usually deployed in highly distributed and mobile situations and tend to be used in critical applications. To reduce financial and/or human risks, rigorous verification of such applets is often required to guarantee the intended behaviour of the system to which these applets belong.

The relevance of formal specification and verification methods for Java Card applications is reflected by the large amount of work in this area. A typical example of this is the Mondex case study [19]; an electronic purse application specified in the context of the Grand Challenge on Verified Software. The Mondex specification ensures some important properties of the system, such as that no money may be created by the application.

The Java Modelling Language (JML) [21,23] is a language designed to specify Java programs in detail. Software developers can use JML to add specifications in accordance with the Design by Contract [29] principles by means of assertions, such as method preconditions and postconditions and class invariants. JML annotations can be automatically translated into runtime assertion checking code by JMLc [10,7], the JML compiler. JMLc produces Java executable bytecode supposed to run on any Java virtual machine where the JML runtime classes are available.

* Corresponding author at: DIETInf - Instituto Federal do Rio Grande do Norte, Caixa Postal 1559, 59.015-000, Natal, RN, Brazil. Tel.: +55 84 9911 1931.
E-mail addresses: umberto@dimap.ufrn.br (U.S. da Costa), anamaria@dimap.ufrn.br (A.M. Moreira), mam@dimap.ufrn.br (M.A. Musicante), placido.neto@ifrn.edu.br (P.A. Souza Neto).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.03.003

The usefulness of Design by Contract in general and JML for Java is already well established, as presented in [23]. Due to the critical nature of smart card applications, runtime verifications associated to Design by Contract could contribute to the development of more robust code, e.g., by dealing with exceptional behaviour. However, JML and JMLc are *not* supported by the Java Card virtual machine. The input programming language accepted by the Java Card virtual machine has been restricted to cope with the restrictions imposed by the target devices where most of Java Card applets run. A consequence of this restriction is that Java Card cannot benefit from JML specification and verification tools in order to improve the safety of its applets at runtime.

The motivation for our work is that, although the Java Card virtual machine is not able to deal with the code produced from a full JML specification, the safety of Java Card applets can be improved at least by a subset of JML. Such a subset can be defined in order to avoid all those features that are not supported by the Java Card virtual machine. It is necessary to ensure that both data and control structures involved in the specifications as well as the code generated for the verification of these specifications are compliant with the Java Card virtual machine.

In this context, the main contributions of this work are:

- The proposal of JCML, a restricted version of JML as a Java Card specification language.
- The design and implementation of a compiler for this language. Unlike the original JML compiler, our implementation focuses on the generation of concise and efficient code. To achieve this goal: (1) some optimization techniques are used for the generation of the Java Card code that will be run on-card, and (2) some verifications cannot be dealt with completely, and, if needed, have to be performed statically or in an off-card testing environment.
- Two case studies where the size and execution times of the JCML-generated programs are compared to those programs generated by the standard JML compiler. The size of the generated code is the main restriction when considering constrained devices after compatibility with the Virtual Machine. Execution times come just afterwards, as card processors are not as powerful as desktop ones and it is not acceptable for a card to take significant amounts of time to respond to the user's requests.

This paper is an extended, revised version of [11,12]. Some important, new contributions have been included, such as: (i) a more precise and complete definition of the JCML language; (ii) the implementation of some new verification items and static analysis to improve the verification of non-boolean conditions; (iii) the addition of a new case study (iv) and the inclusion of new experiments and results.

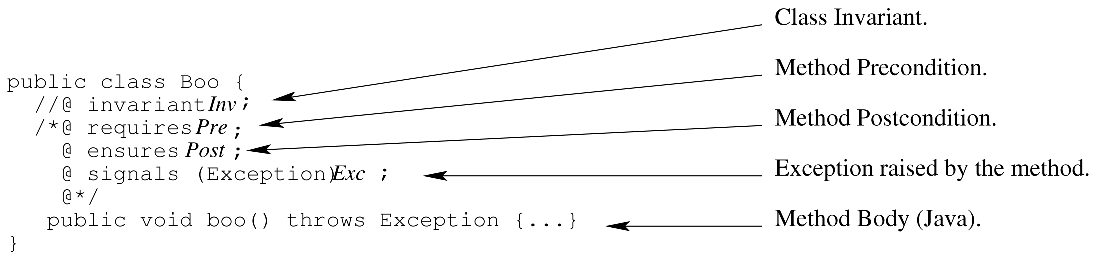
This paper is organized as follows: Section 2 presents Design by Contract and JML. Section 3 introduces the Java Card platform and discusses related limitations and advantages. After that, Section 4 shows how Java Card restrictions affect the definition of JCML and presents JCML in detail through its grammar. Section 5 introduces the implementation of the JCML compiler, including the translation from JCML assertions to Java Card code. Experimental results are shown in Section 6, where we compare the code produced by the JCML compiler and that produced by the JML compiler in terms of size and execution times. Finally, Sections 7 and 8 present some related work and the next steps to be taken to improve JCML and its compiler, as well as our final remarks.

2. Design by contract and JML

Design by Contract [28,29] is a software development method based on the definition of contracts between software units. The method proposes the run-time verification of these contracts. Design by Contract uses logical assertions (preconditions, postconditions and class invariants) to specify contracts. These assertions can be added to the source code of a program and can be dynamically verified.

JML implements these ideas by using model-based specifications with mathematical concepts such as sets and first order logic to specify the contents of each contract condition. However, differently from other model-based specification languages such as Z [40] and B [1], JML uses Java syntax as much as possible in its specifications, adding extra constructs to cope with specification concepts that are not directly expressed in Java, such as first order logic quantifiers.

In JML, assertions are expressed within the Java program by using a special kind of comment which expresses the conditions as first-order logic formulae, as follows:



```
public class Boo {
  //@ invariant Inv ;
  /*@ requires Pre ;
    @ ensures Post ;
    @ signals (Exception) Exc ;
    @*/
   public void boo() throws Exception {...}
}
```

JML supports Design by Contract by means of two styles of specifications: *lightweight* specifications and *heavyweight* specifications. Roughly, *lightweight* specifications may be partial, i.e., specifications where only part of the required

```
1 import ...
2 public class UserAccess {
3
4  public static final byte  MAX_USER_ID_LENGTH = 15;
5
6   //types of users
7   public static final byte  STUDENT = 0;
8   public static final byte  PROFESSOR = 1;
9
10  //different requirements for  different types of users
11  public static final byte  MAX_AREAS = 20;
12  public static final short MAX_CREDITS = 3000;
13  public static final byte  STUDENT_MAX_AREAS = 10;
14  public static final short STUDENT_MAX_CREDITS = 1000;
15
16  //class attributes
17  private /*@ spec_public @*/ byte[] userId;
18  private /*@ spec_public @*/ byte   userType;
19  private /*@ spec_public @*/ byte[] authorizedAreas;
20  private /*@ spec_public @*/ byte   nextArea;
21  private /*@ spec_public @*/ short  printerCredits;
22
```

Fig. 1. Initial part of the `UserAccess` class definition with JML annotations.

properties of a class or method are specified. *Heavyweight* specifications, on the other hand, are supposed to be complete, i.e., if an item is not explicitly stated, the default specification condition for that item is assumed.

JML annotated Java programs may be compiled with the JML compiler into instrumented Java programs. These resulting programs perform the original computation together with the verification steps. For instance, if a precondition p is specified for a method m and, at runtime, m is called in a context where p evaluates to false, the execution of m can be blocked and a warning message be sent to the user. Also, if in spite of all care, a specified class invariant is broken by some method execution, this situation can be immediately detected and made visible to the user. Together with static (type) verification, the dynamic verification of assertions increases confidence in the applications, and so, it should be particularly useful in the smart card domain. However, because of this domain's severe restrictions, a JML-like language was not yet available for it. Up to now, most smart card code is manually programmed and inserted into the card programs, without any kind of formal verification.

One of the goals of JML is to provide a tool that can be used for industrial applications. This means that a precise definition of the features of the language is needed, to be used by application programmers. At the same time, JML is a language under development, allowing several research groups to define and experiment with new features. In order to comply with these two, apparently conflicting, requirements, JML defines six levels, containing different syntactic constructors. These levels [22] are targeted at different classes of users and are described as follows:

Level 0: This level defines the kernel of the language, including those features that any JML implementation must include. It represents the *stable, fundamental* version of the language.
Level 1: The syntactic constructors present in this level include features that can be defined in terms of several level 0 constructors (syntactic sugar), redundant features as well as some features for which static verification is still problematic (including method and constructors calls in assertions).
Level 2: Contains specialized features, to be used by specific kinds of users.
Level 3: This level contains even more exotic and semantically not-well-understood features.
Level C: The features of this level are devoted to the use of JML to annotate Concurrent Java programs.
Level X: Includes experimental features, to be included at other levels in future releases of the framework.

2.1. An example

Let us now present parts of an application named *UserAccess* (Figs. 1–3), to be used by students and staff of a university. The application runs a printing management quota system and also grants access to certain parts of the building. We propose a JML specification for this application. It will be used to present some key features of JML.

The *UserAccess* class includes methods for: defining and returning the user ID (`setID(byte[] m)`, `getID()`), adding rights of access to a location to the user or checking if the user has the right to access a given area (`addArea (byte local_cod)`, `hasAccess(byte local_cod)`, adding or using printer credits (`addCredits(short value)`, `rmCredits(short value)`), checking the balance of printer credits (`short getCredits()`), defining and returning the user type (`setType(byte[] m)`, `getType()`). This class is defined in standard Java code to which JML annotations are added through special comments limited by "`/*@`" and "`@*/`". In lines 17 to 21, the `spec_public` annotations state that these attributes, although private in the context of the Java code, are public in the context of the specification.

Specification of Invariants. Invariants for a class are properties that must hold throughout the execution of every instance of the class. In JML, one can specify static and instance invariants: static invariants are properties over static attributes and

```
23  // no userId may have more than MAX_USER_ID_LENGTH
24  /*@ invariant userId.length <= MAX_USER_ID_LENGTH; @*/
25
26  //every user is either a student or a professor
27  /*@ invariant userType == STUDENT || userType == PROFESSOR; @*/
28
29  // global limits and values
30  /*@ invariant authorizedAreas.length <= MAX_AREAS; @*/
31  /*@ invariant \forall byte a; 0 <= a && a < authorizedAreas.length;
32                authorizedAreas[a] >= 0; @*/
33  /*@ invariant printerCredits >= 0 &&
34                printerCredits <= MAX_CREDITS; @*/
35
36  //restricted limits for students
37  /*@ invariant userType == STUDENT ==>
38                authorizedAreas.length <= STUDENT_MAX_AREAS; @*/
39
40   /*@ invariant userType == STUDENT ==> printerCredits <=
41                                  STUDENT_MAX_CREDITS; @*/
...
```

Fig. 2. `UserAccess` class invariants.

```
1 /*@ requires value >= 0 &&
2           (value + getCredits()) <= MAX_CREDITS &&
3           (userType == STUDENT ==>
4               (value + getCredits()) <=STUDENT_MAX_CREDITS);
5     ensures printerCredits >= value;
6 @*/
7   public void addCredits(short value) {
8        printerCredits += value;
9    }
10
11  public short getCredits(){
12       return printerCredits;
13  }
```

Fig. 3. `addCredits` and `getCredits` Methods.

instance invariants deal with instance fields. Instance methods must comply with both static and instance invariants. The only variables allowed to appear in JML invariants are class attributes. In runtime verification, invariants are supposed to be checked before and after each method is called.

The *UserAccess* invariant (Fig. 2, lines 23–41) defines: `(i)` that no userId may have more than MAX_USER_ID_LENGTH characters (line 24); `(ii)` that every user is either a student or a professor (line 27); `(iii)` global limits and values for the number of authorized areas and printer credits per user (lines 30, 33 and 34); `(iv)` restricted limits for students (lines 37 to 41); `(v)` that area codes are natural numbers (lines 31 and 32).

UserAccess Methods Specification. Let us now see how a method may be annotated with JML. The `addCredits` method is used to add printer credits to the user. This method, shown in Fig. 3, has one parameter, called `value`, corresponding to the amount to be credited. The specification requires that the value to be credited is non-negative (line 1), and that the resulting credit balance is not greater than the allowed limit for the user. Notice that the specification uses the method `getCredits()`, which provides the current balance stored in the card (lines 2 to 4). The specification also states (at line 5) that, at the end of the method execution, the available credit is at least the amount which was credited. One important thing to notice here is that the code that implements the method (lines 7 to 9) does not take care of these restrictions. It will just execute the assignment (line 8), independently of whether the assertions hold or not, leaving the responsibility of verifying the satisfaction of the precondition to the user (or to the verification code generated by the specification compiler).

2.2. Static and dynamic verification of contracts

The verification of contracts consists in checking that the assertions that define the contract hold. This activity can be performed at compile time as well as during the execution of the program. In our context, static and dynamic verifications are complementary and are used together. The use of static analysis can achieve a more general coverage of the program's properties than the use of dynamic techniques. This is justified by the fact that static verification can explore execution paths that are rarely activated in practice. Static analysis also can achieve a better coverage by using abstractions on the program and data-flow analysis performed over source or object code. On the other hand, since static analysis deals only with the (incomplete) information known at compile-time, some verifications are hard or even impossible to be performed, such as deciding when different object references are aliases for a same object, verifying conditions over recursive data structures or even the existence of certain classes of possibly non-terminating loops [2]. Such kind of verification can be done at runtime when the state of the program is completely known. Runtime verification also copes more easily with problems such as the one dealt with by [26] in a static verification context where it is difficult to guarantee invariant validity on an object oriented context.

Static analysis is usually employed by formal verifiers that perform proofs of correctness given a specification or a model. So, whenever the existence of an error is not conclusive such verifiers issue "false positives", that is, errors that do not occur in practice. This reduces the effectiveness of the produced warnings. In order to reduce false positives, the code can be annotated with conditions under which the execution must be considered.

Dynamic analysis can be part of automated testing tools that interleave the execution of the original program with testing routines. In practice, calls to the original routines are replaced with calls to versions where the contract verification is performed (and error messages are generated in case of non-compliance). So, dynamic analysis has access to all the information regarding the execution environment and can issue detailed error reports for given inputs, without false positives. Since program execution is inherent to dynamic analysis, such an approach increases the size and running time of programs. Testing is intended to reveal the presence of errors, not to prove correctness. Therefore, the effectiveness of tests depends on the selection of inputs and there are no guarantees about activating faulty execution paths. On the other hand, automated testing requires less specialized knowledge and can scale to larger designs than the static approaches [3].

The combination of static and dynamic analysis seems to be the best choice in the case of contract verification, each one complementing the other. Static analysis can be used to guide the dynamic testing by reducing the number of conditions to be observed during runtime and improve performance, whereas dynamic testing can discover errors that arise in practice but that are difficult to find by static approaches. In [3,13,35] different interactions between static and dynamic analyses are shown.

3. Java Card

Java Card is a Java platform designed for resource-constrained devices. Due to the nature of those devices, the platform is quite limited. Memory restrictions are crucial for the design of Java Card applications: typical smart cards have as low as 12KB RAM, 374KB ROM and 144KB EEPROM. The read-only memory is used to store the Java Card Virtual Machine (JCVM), a severely downsized version of the JVM. The main differences between JVM and JCVM standards are the exclusion of some important JVM features such as many primitive types, dynamic classes and threads.

Java Card applications are called applets. In order to run one of these applets in a Java Card device, one must: (1) write the Java Card applet; (2) compile the applet; (3) convert the binary classes into a converted applet CAP file; (4) install the CAP file on the card; (5) run the applet. Compliance problems are detected by an off-card component of the JCVM [9], the converter, before the applet is installed on the card.

Because of these restrictions, a typical Java Card application will be very limited, and only some basic functionalities will be provided on-card. Most of the more heavy processing is executed on the so-called *host side*, the program that runs on the terminal to which the card is temporarily connected. However, for safety reasons, some card data may not be seen by the host application. Such sensitive data must be manipulated on the card, safely and correctly. This is one of the advantages of smart cards with respect to magnetic strip cards: their on-card code may be used to ensure the safety and correctness of data apart from the host application. This is where tools for a rigorous specification and verification of on-card data and functionalities become necessary.

Java Card API. The Java Card API specification [30,20] defines a small subset of the traditional Java programming language API which is supported by JCVM. On top of this, the Java Card framework defines its own set of core classes that specifically support Java Card applications. The main packages are described below:

1. *javacard.framework*: Defines the interfaces, classes, and exceptions for services that compose the core Java Card Framework.
2. *javacard.security*: The Java Card specification defines a robust security API that includes various types of private and public keys and algorithms, e.g., *Key*, *PrivateKey*, *PublicKey* interfaces, and, *Checksum*, *KeyAgreement*, *Signature* classes;
3. *java.io*: Defines one exception class: the *IOException* class. None of the other traditional java.io classes are included;
4. *java.lang*: Defines *Object* and *Throwable* classes that are defined in the J2SE framework, the *Exception* class, various runtime exceptions, and the *CardException* class. None of the other traditional java.lang classes are included.
5. *java.rmi* : Defines the Remote interface and the *RemoteException* class. None of the traditional java.rmi classes are included.

*Java Card and the **Mondex** electronic purse system.* Mondex is a smart card electronic purse system. It was defined around 1996 and is part of the MasterCard suite of products. Mondex has been formally verified, and it is a key contribution to the *Grand Challenge in Verified Software* [18] repository. A number of works address the Mondex System [17,4,41,15,34]. The work on [34] describes a Java Card implementation of the Mondex protocol, obtained by refinement of a Z specification into a JML specification and statically verified using the KeY system [6].

We used this JML Mondex specification, as described in [38], as a *real world* case study to test our approach, complementing previous simpler case studies that had previously been run.

Some results of this case study are presented in Section 6.

(0) *jcml-compilation-unit* ::= [*package-definition*]
(*import-definition*)* (*type-definition*)*

(1) *import-definition* ::= [**model**] **import** *name-star*

(2) *type-definition* ::= *class-definition* | *interface-definition* | **;**

(3) *class-definition* ::= *modifiers* **class** *ident* [*class-extends-clause*]
[*implements-clause*] *class-block*

(4) *interface-definition* ::= *modifiers* **interface** *ident*
[*interface-extends*] *class-block*

(5) *class-extends-clause* ::= **extends** *name* [**weakly**]

(6) *implements-clause* ::= **implements** *name-weakly-list*

(7) *interface-extends* ::= **extends** *name-weakly-list*

(8) *name-weakly-list* ::= (*name* [**weakly**])+

(9) *modifiers* ::= *modifier**

(10) *modifier* ::= **public** | **protected** | **private** | **abstract** | **static**
| **final** | *jcml-modifier*

(11) *jcml-modifier* ::= **spec_public** | **spec_protected** | **model**
| **ghost** | **pure** | **instance** | **helper**
| **uninitialized**
| **non_null** | **nullable** | **nullable_by_default**

Fig. 4. JCML grammar — *(part 1)*.

4. Java Card Modelling Language (JCML)

JML is an extension of Java that includes specification constructs. These constructs can contain Java code that is used only when the JML specification is verified. Likewise, JCML specifications are Java Card programs annotated with specification constructs. The specification part of a JCML program is defined using a special kind of comment. Java Card constructs defined within the annotations are treated by our JCML compiler and can be used in the pre/postconditions and invariants.

To specify and verify Java Card applications, only those constructs that are Java Card-compliant can be used. Such a restriction leads to two different, but related, requirements for a Java Card-compliant variant of JML: (1) Java Card constructs, instead of Java constructs, are allowed in the specification, and (2) the additional runtime verification code, which is generated by the tool, has to be formed by Java Card constructs. As a consequence, not only must the specification language be Java Card compliant, but the verification code generator must also be designed to generate strictly Java Card compliant code to perform runtime verification. Beside these restrictions, time and memory consumption have to be taken into account on the definition of the modelling language and the corresponding verification code. The design of such a specification language has to consider the trade-off between expressiveness and feasibility.

With these requirements in mind, in [11,12] we first presented JCML: the *Java Card Modelling Language*. These previous works only present an informal overview of the language. In this paper we give JCML a more precise presentation of the syntax and compilation process of the language.

JCML was designed to be as close as possible to JML. More precisely, almost all of the JML fundamental level (level 0) constructs are included in JCML (and some features of other levels). This means that many JML constructs are present in JCML even if their runtime verification on a smart card platform is not feasible with current technology. In this case, the construct may be present in the language only as a specification tool, but no verification code is generated for it.

To fulfil the goal of staying as close as possible to JML, the grammatical definition of JCML was derived from the grammar for JML in [22]. The grammatical definition of JCML is given next, stressing on the explanation of those aspects of JCML that differ from JML. The EBNF (*Extended Backus-Naur Form*) for JCML is shown in Figs. 4–11. Notice that the grammar for JCML includes a Java Card-compliant subset of Java, in the same way as JML includes Java. For space reasons, our presentation focuses on the specification-related parts of JCML. It includes some Java Card constructors, when necessary, in order to clarify the explanation. We use the following convention: non-terminal symbols are written in *this italic font* while non-terminals symbols are written in **bold roman** characters.

We start with the initial symbol, *jcml-compilation-unit* (Fig. 4, rule 0). A JCML compilation unit is a Java card construct which defines zero or more types (classes or interfaces) and import declarations; the whole, possibly grouped inside a package. Comparing with the original JML grammar, two syntactic classes have been removed from the compilation unit rule: MultiJava constructs (our JCML compiler does not extend the MultiJava Compiler, but the standard Java one) and JML specification refinement constructs. Refinement constructs are not included in the present version of the tool since they are

(12) *class-block* ::= **{** *(field)** **}**

(13) *field* ::= *member-decl* | *jcml-declaration* | *class-initializer-decl*

(14) *member-decl* ::= *method-decl* | *variable-definition*
| *class-definition* | *interface-definition*

(15) *jcml-declaration* ::= *modifiers invariant*
| *modifiers history-constraint*
| *modifiers represents-decl*

(16) *class-initializer-decl* ::= [**static**] *compound-statement*
| *jcml-method-specification* [**static**] *compound-statement*
| *jcml-method-specification* [**static_initializer**]
| *jcml-method-specification* [**initializer**]

(17) *method-decl* ::= *[jcml-method-specification]*
modifiers method-head method-body

(18) *variable-definition* ::= *modifiers variable-decls*

Fig. 5. JCML grammar — *(part 2)*.

(19) *invariant* ::= *invariant-keyword predicate* **;**

(20) *invariant-keyword* ::= **invariant** | **invariant-redundantly**

(21) *history-constraint* ::= *constraint-keyword predicate*
[**for** *constrained-list*] **;**

(22) *constraint-keyword* ::= **constraint** | **constraint-redundantly**

(23) *constrained-list* ::= *method-name-list* | \everything

(24) *represents-decl* ::= *represents-keyword store-ref-expression*
l-arrow-or-eq spec-expression **;**
| *represents-keyword store-ref-expression*
\such_that *predicate* **;**

(25) *represents-keyword* ::= **represents** | **represents-redundantly**

(26) *l-arrow-or-eq* ::= ← | **=**

Fig. 6. JCML grammar — *(part 3)*.

(27) *jcml-method-specification* ::= *specification* | *extending-specification*

(28) *extending-specification* ::= **also** *specification*

(29) *specification* ::= *spec-case-seq [redundant-spec]* | *redundant-spec*

(30) *spec-case-seq* ::= *spec-case* (**also** *spec-case*)*

(31) *spec-case* ::= *lightweight-spec-case* | *heavyweight-spec-case*

Fig. 7. JCML grammar — *(part 4)*.

used at an early moment in the development process, when code is not yet run on the card. We plan to include refinement constructs in a later version of the JCML framework.

The production rule for *package-definition*, used in rule 0, is omitted here as it is the standard Java one and does not lead to any specification. The JCML version of an *import-definition*, is similar to the standard Java Card construct, with the optional addition of a **model** keyword. This keyword is used to declare an import at the specification level (the non-terminal *name-star* corresponds to the usual java name convention as in, e.g., **java.util.***). A *type-definition* (rule 2) may be a class or interface and may lead to JCML specification constructs.

The production rules 3 and 4 in Fig. 4 are identical to the ones of JML and, at this level, are standard Java constructs. However, from this point on, JCML specification constructs begin to show up, such as the specification of a *weak* extension or implementation (rules 5 to 8) which avoid history constraints to be inherited. History constraints (presented in Fig. 6) are global constraints on the behaviour of attributes and methods which are useful to specify global evolution restrictions on attribute values.

A difference between JCML and JML is the set of available modifiers (rules 9 to 11), which may appear in class or interface definitions (rules 3 and 4). The Java modifiers **synchronized**, **transient**, **volatile**, **native**, **strictfp** and **const** are

(32) *lightweight-spec-case* ::= *generic-spec-case*

(33) *generic-spec-case* ::= [*spec-var-decls*] *spec-header* [*generic-spec-body*]
| [*spec-var-decls*] *generic-spec-body*

(34) *spec-header* ::= (*requires-clause*)+

(35) *generic-spec-body* ::= *simple-spec-body* | **{|** *generic-spec-case-seq* **|}**

(36) *simple-spec-body* ::= (*simple-spec-body-clause*)+

(37) *simple-spec-body-clause* ::= *diverges-clause* | *assignable-clause*
| *ensures-clause* | *signals-only-clause*
| *signals-clause* | *captures-clause*

(38) *generic-spec-case-seq* ::= *generic-spec-case* (**also** *generic-spec-case*)*

Fig. 8. JCML grammar — *(part 5).*

(39) *requires-clause* ::= *requires-keyword pred-or-not* **;**

(40) *pred-or-not* ::= *predicate* | \not_specified | \same

(41) *requires-keyword* ::= **requires** | **pre** | **requires_redundantly**
| **pre_redundantly**

(42) *ensures-clause* ::= *ensures-keyword pred-or-not* **;**

(43) *ensures-keyword* ::= **ensures** | **post** | **ensures_redundantly**
| **post_redundantly**

(44) *signals-clause* ::= *signals-keyword* **(***reference-type* [*ident*] **)**
[*pred-or-not*] **;**

(45) *signals-keyword* ::= **signals**| **ensures** | **ensures_redundantly**
| **signals_redundantly**

(46) *signals-only-clause* ::= *signals-only-keyword reference-type*
(**,** *reference-type*)* **;**
| *signals-only-keyword* \nothing ;

(47) *signals-only-keyword* ::= **signals_only** |**signals_only_redundantly**

(48) *diverges-clause* ::= *diverges-keyword pred-or-not* **;**

(49) *diverges-keyword* ::= **diverges** | **diverges_redundantly**

(50) *assignable-clause* ::= *assignable-keyword* (*ident* (, *ident*)*
| *store-ref-keyword*) **;**

(51) *assignable-keyword* ::= **assignable** | **assignable_redundantly**
| **modifiable** | **modifiable_redundantly**
| **modifies** | **modifies_redundantly**

(52) *store-ref-keyword* ::= \nothing | \everything | \not_specified

(53) *captures-clause* ::= *captures-keyword store-ref-list* **;**

(54) *captures-keyword* ::= **captures** | **captures_redundantly**

Fig. 9. JCML grammar — *(part 6).*

not recognized in Java Card, and they are not included in JCML. Similarly, the JML specification modifiers **spec_java_math**, **code_java_math**, **spec_safe_math**, **code_safe_math**, **spec_bigint_math** and **code_bigint_math** are not part of JCML as they do not have a precise definition in the JML Reference Manual (they are classified as level 2 JML). The JML modifiers that are allowed in JCML (rule 11), have a variety of uses, such as changing visibility or restricting the possible values of variables or behaviours of methods.

In Fig. 5, we present the main components of a class or interface definition: A *class-block* (rule 12) is composed by zero or more *fields*. Each field (rules 13 to 18) may be a method declaration (rule 17), a variable, class or interface definition (defined by rules 18, 3 and 4, respectively) or a class initialization and specification (*class-initializer-decl*, rule 16) or a *jcml-declaration* (rule 15).

Rule 17 defines the usual structure for a Java method, with the addition of an optional *jcml-method-specification* (detailed in Fig. 7), and the extra *jcml-modifiers* which, as already seen (in Fig. 4), are included as part of the *modifiers* non-terminal.

```
(55) heavyweight-spec-case ::= behavior-spec-case
                             | exceptional-behavior-spec-case
                             | normal-behavior-spec-case
(56) behavior-spec-case ::= [ privacy ] behavior-keyword generic-spec-case
(57) behavior-keyword ::= behavior | behaviour
(58) normal-behavior-spec-case ::=
               [privacy] normal-behavior-keyword normal-spec-case
(59) normal-behavior-keyword ::= normal_behavior | normal_behaviour
(60) normal-spec-case ::= generic-spec-case
(61) exceptional-behavior-spec-case ::=
          [privacy] exceptional-behavior-keyword exceptional-spec-case
(62) exceptional-behavior-keyword ::= exceptional_behavior
                                    | exceptional_behaviour
(63) exceptional-spec-case ::= generic-spec-case
(64) privacy ::= public | protected | private
```

Fig. 10. JCML grammar — *(part 7)*.

```
(65) jcml-primary ::= result-expression | old-expression | fresh-expression
                    | nonnullelements-expression | informal-expression
                    | typeof-expression | elemtype-expression
                    | type-expression | spec-quantified-expr
(66) result-expression ::= \result
(67) old-expression ::= \old (spec-expression [, ident])
                      | \pre ( spec-expression )
(68) fresh-expression ::= \fresh ( spec-expression-list )
(69) nonnullelements-expression ::= \nonnullelements ( spec-expression )
(70) typeof-expression ::= \typeof ( spec-expression )
(71) elemtype-expression ::= \elemtype ( spec-expression )
(72) type-expression ::= \type ( type )
(73) spec-quantified-expr ::= ( quantifier quantified-var-decls ; [[ predicate ] ; ]
                              spec-expression )
(74) quantifier ::= \forall | \exists | \max | \min |
                  \num_of | \product | \sum
```

Fig. 11. JCML grammar — *(part 8)*.

A *method-head* is similar to its corresponding Java definition, with the exception of the possibility of modifying parameters with the information **nullable** or **non_null**. A *method-body* is a compound statement. Finally, a *variable-definition* is also a regular Java construct, possibly preceded by Java and JCML modifiers. There are two main restrictions with respect to a JML *variable-definition*: data types are restricted to Java Card, and so, only one-dimensional arrays are allowed, for instance, and JML data groups, used for extra access control, have not been included in JCML (but may be included in the future).

Fig. 6 defines the different JCML specification constructs that may appear globally in a class or interface declaration: the *invariant* (rules 19 and 20), stating properties that must always hold, the *history constraints* (rules 21 to 23), which constrain how variables may evolve trough method execution, and the *represents-decl* (rules 24 to 26), which relates model and concrete variables. A *store-ref-expression* identifies a set of memory locations (as in **this**, for instance). *predicate* and *spec-expression* are Java Card expressions extended with JML level 0 predicate constructs such as quantifiers and commands for type retrieval.

The *redundantly* versions of each of the constructs in Fig. 6 are used to state that the corresponding property is a logical consequence of its counterpart. Some JML constructs are not present in JCML: *initially-clause* (an extra predicate to be satisfied by constructors), *monitors-clause*, *readable-if-clause* and *writable-if-clause* (extra access control predicates) and *axiom-clause* (meant to provide provers with additional axioms).

The rules 27 to 31 in Fig. 7 are the same as those found in JML, and define the specification of methods. Specification cases (rule 31) are classified in lightweight or heavyweight, and each specification case may extend previously stated ones by using the keyword **also**. For generality, JCML grammar preserves heavyweight specifications but, due to the restricted platform for which it is being developed, only *lightweight* specifications should be verified, for a complete heavyweight specification verification would in general be too big to run in a smart card.

In Figs. 8 and 9 the rules defining lightweight specification cases are presented, while heavyweight specification cases are presented in Fig. 10. As in JML, stating explicitly redundant specifications is allowed in JCML, but, as explained later in this paper, our JCML verification code generator assumes that this redundancy is to be checked statically, reducing in this way the amount of verification code to be run on the card. We thus do not detail here the redundancy constructions which are the same as JML's. Finally, JCML does not include model specifications, which are an advanced feature of JML (level 2, Section 2), and focuses on development steps which are not related to runtime verification, the focus of our work.

As seen in rules 32 to 38 in Fig. 8, a lightweight specification case (*lightweight-spec-case*) contains an optional declaration part, where specification variables may be defined, an optional series of preconditions (**requires** clauses — rules 34 and 39 to 41), and a specification body (rules 35 to 38). JML specification clauses which were not compatible with Java Card were eliminated: **when-clause** (concurrency related), **working-space-clause** and **duration-clause** (which use the **long** integer to specify heap space or time consumption, respectively). The preserved clauses are the ones in rule 37: **diverges-clause** (specifies that a method may not return — rules 48 and 49), **assignable** (declares which variables and objects may have there value changed during method execution — rules 50 to 52), **captures** (declares when an object may be captured during a method call — rules 54 and 55), **ensures** (specifies normal execution postconditions — rules 42 and 43), **signals-only** and **signals** (indicate which exceptions may be thrown by the method — rules 44 to 47).

As far as syntax is concerned, heavyweight specification cases (rules 55 to 63 in Fig. 10) are similar to lightweight specification cases (it all reduces to *generic-spec-case*), with the special behaviour keywords (rules 57, 59 and 62) and the possibility to override code visibility rules with extra privacy indicators (rules 56, 58, 61 and 63). The only difference here with respect to JML is the removal of the **code** keyword, which is not part of JML level 0, and allows to block inheritance of a method specification in case the method is overridden.

Finally, in Fig. 11 we present the JCML specification specific parts of a predicate expression, which may appear in many places of a JCML annotated Java Card program (e.g., invariants and constraints in Fig. 6). These constructs are those of JML level 0. A predicate may then refer to the return value of a method (*result-expression*, rule 66), to the value of parameters and variables before a method's execution (*old-expression*, rule 67), to a type (*typeof-expression*, *elemtype* and *type*, rules 70 to 72), a quantified expression (*spec-quantified-expr*, rules 73 and 74), or to the fact that objects in an expression are fresh, i.e., newly allocated (*fresh-expression*, rule 68). This last one, although classified as JML level 0, is not reasonable in the context of Java Card applications where objects are usually allocated once and reused as much as possible throughout execution. It has not been implemented in JCML.

5. The JCML compiler

The current version of the JCML compiler generates Java Card-compliant verification code for lightweight specifications. It implements invariant and condition verifications. The organization of the generated code preserves the structure of the original Java Card program, adding the verification code to it. The focus of the implementation is the generation of code suitable to be run on very restrictive devices. Because of this, the JCML compiler had to be developed from scratch.

5.1. General structure of the generated code

The JCML compiler originally used the *wrapper* approach proposed in [10]. In this approach, some auxiliary methods are generated for the verification of preconditions, postconditions and invariants. These auxiliary methods rise exceptions when the assertions are violated. The original method being verified is renamed and made private. Finally, a *wrapper method* with the same name and signature as the original method it wraps is defined. In its body, it checks the preconditions and invariants and then executes the original method. After that, the wrapper checks the invariant and any specified postconditions.

The experiments in [11,12], updated in Section 6, show however that this approach presents a drawback in execution times, which is an important feature of a Java Card application. For this reason the current version of the compiler allows the user to specify, through a compiler directive, among four different code generation patterns, which the user may choose depending on the available memory and execution speed requirements:

1. **-wrapper:** Uses the wrapper approach, as presented above.
2. **-inLineMethod:** Method inlining is used for the original, user methods. Instead of creating new, private methods, the body of each method is inlined into its wrapper (i.e, the method call inside the wrapper is substituted by the method's body). This is the default option in our implementation.
3. **-inLinePrePost:** Inlining of the original methods, and the ones which carry out pre- and postcondition verification. Invariant-checking methods are still generated.
4. **-inLineAll:** All the methods for checking preconditions, postconditions and invariants, as well as the original Java method are inlined into the wrapper.

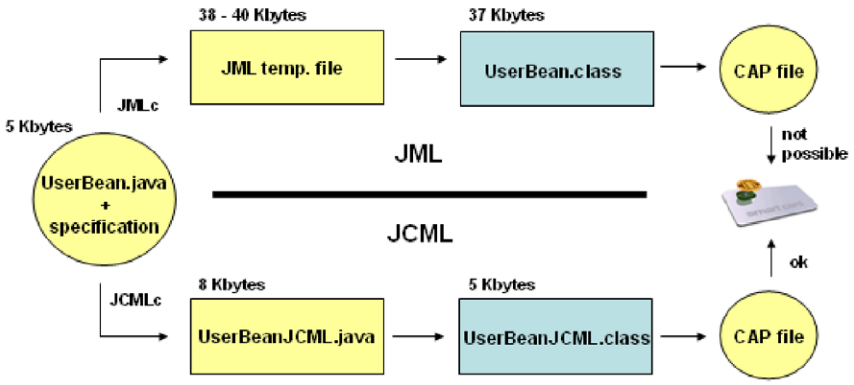


Fig. 12. JCMLc x JMLc.

```
1 private void checkInv$ClassName$() {
2  try{
3    if (!(P1)) {
4     ISOException.throwIt(InvariantException.SW_INVARIANT_ERROR);}
5       ...
6    if (!(Pn)) {
7     ISOException.throwIt(InvariantException.SW_INVARIANT_ERROR);}
8   }catch(Exception e){
9     ISOException.throwIt(InvariantException.SW_INVARIANT_ERROR);
10  }
11}
12
13 private void checkPre$MethodName$(T1 a1,..., Tk ak) {
14  try{
15   if (!(Q1)) {
16    ISOException.throwIt(RequiresException.SW_REQUIRES_ERROR);}
17      ...
18   if (!(Qm)) {
19    ISOException.throwIt(RequiresException.SW_REQUIRES_ERROR);}
20  }catch(Exception e){
21    ISOException.throwIt(RequiresException.SW_REQUIRES_ERROR);
22  }
23}
```

Fig. 13. Java Card methods to check invariants and preconditions.

Our compiler reads a source file *<name>*`.java` and produces a Java file named *<name>*`JCML.java`. The generated Java program is similar to the source file, with the addition of the assertion-checking methods. It can be compiled with a standard Java Card compiler in order to generate the executable class, and finally converted into a CAP file to be run on a smart card. This process, shown in Fig. 12, is similar to the one implemented by JMLc, the JML compiler. Notice that the file generated from the JML specification cannot be run on-card, even if the original annotated code is Java Card compliant. This happens because the verification code generated by JMLc uses libraries and types which are not in the Java Card platform.

In Fig. 13 we present the structure of the auxiliary methods that check invariants and preconditions. Each assertion is defined by a predicate `Pi`. The condition verified by the auxiliary method `checkInv$ClassName$` (lines 1–11) corresponds to the conjunction of all the predicates for the invariants for the class `ClassName`. This method does not take parameters, since invariants are defined over global variables. If any of the predicates `Pi` evaluates to *false*, an `InvariantException` is signalled.

The `checkPre$MethodName$(T1 a1,. . . , Tn an)` method (defined in Fig. 13, lines 13–23) performs the verification of a precondition `Q1 & . . . & Qm`. This method has the same parameters as the method whose precondition is being verified. As in the case of invariants, an exception will be raised if the precondition evaluates to false. The case of postconditions is analogous.

5.2. *Implemented semantics of boolean expressions*

In recent versions of the JML compiler [22, §2.7], the semantics of the evaluation of undefined expressions has been modified. The original semantics [10,21] tried to mimic the mathematical logic interpretation of formulæ. This was done by approximating their truth value, by considering both *angelic* and *demonic* behaviours of undefined expressions, using contextual information. The new versions of JML Runtime Assertion Checker (RAC) incorporate a more traditional approach to the validation of boolean expressions. The goal of the new approach is to match the semantics of undefined expressions as defined for the Java programming language meaning that "an assertion is considered valid if and only if its evaluation

```
private void checkInv$UserAccessJCML$() throws InvariantException{
 try{
  if (!(userId.length <=MAX_USER_ID_LENGTH ))              //(i)
     ISOException.throwIt(InvariantException.SW_INVARIANT_ERROR);
  if (!(userType == STUDENT || userType == PROFESSOR ))   //(ii)
     ISOException.throwIt(InvariantException.SW_INVARIANT_ERROR);
  if (!(authorizedAreas.length <=MAX_AREAS ))            //(iii)
     ISOException.throwIt(InvariantException.SW_INVARIANT_ERROR);
  for (byte a = 0; a < authorizedAreas.length ;a++){
    if (!(authorizedAreas [a]>=0 ))                        //(v)
     ISOException.throwIt(InvariantException.SW_INVARIANT_ERROR);
  }
  if (!(printerCredits >=0 &&
            printerCredits <=MAX_CREDITS ))              //(iii)
     ISOException.throwIt(InvariantException.SW_INVARIANT_ERROR);
  if (!(!(userType == STUDENT ) ||
        ( authorizedAreas.length <=STUDENT_MAX_AREAS )))  //(iv)
     ISOException.throwIt(InvariantException.SW_INVARIANT_ERROR);
  if (!(!(userType == STUDENT ) ||
        ( printerCredits <=STUDENT_MAX_CREDITS )))        //(iv)
     ISOException.throwIt(InvariantException.SW_INVARIANT_ERROR);
 }catch(Exception e){
   ISOException.throwIt(InvariantException.SW_INVARIANT_ERROR);}
}
```

Fig. 14. UserAccessJCML Check Invariant Method.

(terminates and) results in true without raising an exception'' [8, (*sic*)]. As shown in [8], the new semantics are not only more familiar to developers but also their implementation is simpler, yielding to more concise and efficient code. This kind of interpretation is present in our tool since its first version [36]. The very restrictive nature of our compiler's requirements imposed to adopt these semantics; the size of the code generated to check (possibly undefined) conditions in early versions of the JML compiler was one of our first concerns. Our approach is shown in lines 8–10 and 20–22 of Fig. 13, where we define the treatment of exceptions (of any kind) raised during the evaluation of conditions. These lines implement a *strong validity semantics* for conditions, where undefined expressions provoke the rise of an exception. Our implementation is even simpler than that described in [8], since we have restricted the assertion language in order to (i) have a very simple form of quantified expressions (see Section 5.4) and (ii) rule out all forms of informally-defined assertions.

5.3. Example

The *UserAccess* example presented in the presentation of JML in Section 2.1, is actually JCML compatible. It is used here to demonstrate how runtime verification code is generated.

Verification of invariants. As class invariants are properties that must hold throughout the execution of every instance of the class, JCML invariants are checked before and after each method is called. The JCML compiler generates a separate invariant method for each kind of invariant (class or instance) found in the source code.

The JCML compiler translates the invariant expressions into a private invariant-checking method that raises an `InvariantException` when one of the conditions is broken. The generated code is shown in Fig. 14, where manual comments have been included to associate each verification to the items above. The verification code for the forall quantifier, item `(v)`, includes a for-loop that will be explained in Section 5.4.

UserAccess Methods Verification. Let us now see how a JCML-annotated method is dealt with. As explained in Section 2.1, the `addCredits` specification requires that the resulting credit balance is not greater than the allowed limit for the user. The `addCredits` wrapper method (Fig. 15), generated by JCML, wraps the original method call in a try-catch block that (i) checks the invariant and precondition; (ii) calls the original method and (iii) checks the invariant and postcondition. Fig. 16 shows the generated code for `checkPre$addCredits$`.

5.4. Supporting non-Java operators

JCML includes, in its assertions, some operators that are not primitive in Java (nor in Java Card). These operators are a logical implication operator (==>) as well as universal and existential quantifiers. The JCML compiler generates the implementation of these operators in Java Card, as follows:

Modelling implications. Formulae of the type `A ==> B` found in a specification generate verification conditions corresponding to their equivalent: `!A || B`.

Modelling quantifiers. Consider the JCML quantified expression:

```
\forall short i,j; 0 <= i && i < j && j < 10; a[i] < a[j];
```

```
public void addCredits(short value) {
  try{
      checkInv$UserAccessJCML$();
      checkPre$addCredits$(value);

      addCredits$original$(value); // Call the original method

      checkPost$addCredits$(value);
      checkInv$UserAccessJCML$();
   }catch (InvariantException invEx) {
       ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
   }catch (RequiresException reqEx) {
       ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
   }catch (EnsuresException ensEx) {
       ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
   }
}
```

Fig. 15. Generated addCredits Methods (wrapper).

```
private void checkPre$addCredits$( short value)
         throws RequiresException{
  try{
     if(!(value >=0 && (value + getCredits ())<=MAX_CREDITS &&
            (!(userType == STUDENT ) ||
            ( (value + getCredits ())<=STUDENT_MAX_CREDITS ))))
       ISOException.throwIt(RequiresException.SW_REQUIRES_ERROR);
  }catch(Exception e){
      ISOException.throwIt(RequiresException.SW_REQUIRES_ERROR);
  }
}
```

Fig. 16. Generated method for precondition verification.

which follows the JCML grammar rule:

(73) *spec-quantified-expr* ::= *quantifier quantified-var-decls* **;** [*predicate* **;**] *spec-expression*

This expression uses the universal quantifier to specify that the vector a is sorted at indexes between 0 and 9. According to the JML Reference Manual [22], in the absence of a range predicate, the quantified expression must be evaluated over every value of the type of the quantified variable. For instance, in the example, the verifier would check all the possible values of i and j, from *MinShort* to *MaxShort*. Evaluating this kind of expression can be very problematic in the context of Java Card: even for short-based types, the time required for on-card verification can be unacceptable.

The JCML compiler tackles this problem by using static analysis to reduce the search space. The algorithm processes the conditions in a quantifier, in order to restrict the range of values assumed by each variable. Initially, our algorithm assumes the lower and upper bounds of the type, and proceeds in two steps: (1) the definition of tighter bounds for each variable and (2) the code generation.

The first step is a traversal of the quantifier's predicate. In the example predicate given above, we have the following sequence of upper and lower bounds for its variables:

1. We begin with $\mathtt{MinShort} \le \mathtt{i} \le \mathtt{MaxShort}$ and $\mathtt{MinShort} \le \mathtt{j} \le \mathtt{MaxShort}$.
2. From the equation $0 \le \mathtt{i}$ we can define a new lower bound for i. Now we have $0 \le \mathtt{i} \le \mathtt{MaxShort}$ and $\mathtt{MinShort} \le \mathtt{j} \le \mathtt{MaxShort}$.
3. From the equation $\mathtt{i} < \mathtt{j}$ we can define new bounds for both i and j. Now we have $0 \le \mathtt{i} \le \mathtt{j} - 1$ and $\mathtt{i} + 1 \le \mathtt{j} \le \mathtt{MaxShort}$.
4. Finally, from the condition $\mathtt{j} < 10$ we can define a new upper bound for j. Now we have $0 \le \mathtt{i} \le \mathtt{j}-1$ and $\mathtt{i}+1 \le \mathtt{j} \le 9$.

The second step generates code to verify the specification in accordance with the bounds defined in the first step. For each variable, a (nested) for-loop is generated using the calculated upper and lower bounds. The order in which the variables are defined is relevant. For instance, in the given example, the bounds for i must be absolute. This means that the expression $\mathtt{j} - 1$ must be replaced by its maximum possible value 8. The bounds for j can refer to i. Our algorithm produces the following code:

```
private void checkMethod() {
 for (i = 0; i <= 8; i = i + 1) {
   for (j = i+1; j <= 9; j = j + 1) {
           if (!(a[i] < a[j])) throws Exception; }}}
```

The range predicate presented above is a conjunction of conditions (that can be part of a larger, disjunctive expression). Our algorithm will suppose that predicates in the quantifier expressions are given in ***disjunctive normal form***. For each one of the (conjunctive) components of a disjunctive clause, the algorithm will produce a nested for-loop. The loops generated for each conjunctive clause will be placed in a sequence. For instance, the code generated for a disjunctive clause $\forall i, j.P \vee Q \vee R$ will have the following structure:

```
for (i= ...)
  for (j = ...)
    try {if(!P) throws Exception;
    } catch {
    try {if(!Q) throws Exception;
    } catch {
    if(!R) throws Exception;
    }
```

Even using our algorithm, this number of operations can be too high to be run on-card. Our JCML compiler issues a warning when such a situation arises. The user can enable the code generation for quantifiers, for instance to be used during the application test phase. This code will not be generated for the production, on-card version of the applet.

The treatment of existential quantifiers uses the equivalence $\exists x.P(x) \equiv \neg\forall x.\neg(P(x))$, yielding similar results as for the universal quantifier.

5.5. Implementation details

In order to ensure the optimal use of the resources, the JCML compiler complies with the following requirements:

- *No checking methods or calls are generated for empty specifications*. For instance, if a class does not contain an invariant specification, the method to check invariant is not generated.
- *No checking methods or calls are generated for the trivial specification (true)*.
- *If there is no verification to be done for a method call, then, this method is not renamed and a wrapper for it is not defined.*
- *If the specification to be checked is a Java Card expression, then this expression is used as it is, without generating extra evaluation code for it*. This does not happen, for instance, with quantifiers.
- *Compilation flags are used to set the level of verification required*. For instance, no methods are generated for invariant or postcondition when only preconditions are required.
- *If a specification does not hold, an exception is signalled.*
- *If a specification cannot be checked, it is supposed to be false.* (Closed-world assumption.)

The JCML code generator uses the Visitor Pattern [16] to generate Java Card code. The purpose of the Visitor Pattern is to separate the code generation algorithm for each constructor from the actual syntactic structure of the program. This pattern of software allows to change the operation being performed on a data structure without the need of changing the classes that define the data structure.

6. Experiments, optimization and results

Our experiments consider the *UserAccess* example as well as the annotated classes presented in [33,21] and the Mondex JML specification presented in [38,34]. We compare the code produced by the JCML compiler with the code produced by the original JML compiler [10] in terms of the sizes of the generated code, the size of the executable class files and the execution time for methods. Notice that our compiler generates Java Card code, but the code generated by the JML compiler is not Java Card compliant.

The wrapper approach proposed in Section 5 generates methods for preconditions, postconditions, invariants and renames the original method, which becomes internal and private. This situation generates a large number of method calls.

In the following, we explore the inlining optimization technique for the JCML compiler. Method inlining can have the advantage of reducing the processing time, at the cost of possibly increasing the size of the generated code. Due to the restrictive nature of our applications, the use of this technique on the compiler should be carefully studied. So, we devised several versions of the compiler, using inlining to different extents:

JCMLc1: No inlining. This behaviour corresponds to the **-wrapper** option when calling the JCML compiler.
JCMLc2: Inlining is used only for the original methods. This behaviour corresponds to the **-inLineMethod** command-line option.
JCMLc3: Inlining of original methods, pre- and postconditions. It corresponds to the compiler's **-inLinePrePost** option.
JCMLc4: All the methods for checking preconditions, postconditions and invariants, as well as the original java method are inlined into the wrapper. This version is produced by using the **-inLineAll** command-line option.

Table 1 shows the execution times of each method of the *UserAccess* example (the other examples presented similar general behaviour). Tables 2 and 3 present the total size of code generated by each version of our compiler for the *UserAccess* and *Mondex* examples. In all tables, the column **Original** corresponds to the original annotated program and the columns

Table 1
UserAccess — Execution times (in milliseconds).

Method	Original	JCMLc 1	JCMLc 2	JCMLc 3	JCMLc 4	JMLc
setID	0.005	0.127	0.086	0.034	0.007	0.403
getID	0.005	0.031	0.039	0.014	0.006	0.580
setType	0.005	0.083	0.059	0.014	0.006	0.556
getType	0.005	0.049	0.039	0.014	0.006	0.568
addArea	0.028	0.059	0.060	0.037	0.055	0.726
hasAccess	0.010	0.042	0.058	0.022	0.011	0.723
addCredits	0.005	0.062	0.072	0.060	0.031	0.452
rmCredits	0.005	0.061	0.036	0.017	0.010	0.617
getCredits	0.005	0.035	0.016	0.013	0.006	0.605
Sum	0.073	0.553	0.470	0.229	0.140	5.234

Table 2
UserAccess and Mondex — Sizes (in KB).

Method	Original	JCMLc 1	JCMLc 2	JCMLc 3	JCMLc 4	JMLc
UA.java	3.421	10.806	9.814	9.039	36.123	76.300
UA.class	1.938	5.558	4.572	3.473	8.557	28.200
M.java	18.823	47.309	44.508	39.765	88.455	–
M.class	6.204	13.158	14.879	11.123	22.221	–

Table 3
UserAccess Card Memory Requirements (in KB).

Method	Original	JCMLc 1	JCMLc 2	JCMLc 3	JCMLc 4
UA.cap + *.exp*	2.99	4.24	3.98	3.71	6.72
Memory increase (%)	–	42	33	24	125

JCMLc1 to **JCMLc4** correspond to each version of the program, as generated by our JCML compiler, using the optimization levels described above. Finally, the column **JMLc** corresponds to the code generated by JMLc.

The last row on Table 1 adds up the values in each column to provide a rough idea of general time consumption of each approach. Notice that the inlining of all the generated methods **JCMLc4** resulted in execution times that are comparable to the ones without verification code, even with one quantifier in the invariant that is repeatedly checked. Execution times in Table 1 are the average of CPU times in 10 (ten) executions, given in milliseconds. The experiment was run on a MacBook Core2Duo 2.4GHz with 2GB RAM. The data were collected using the *Profiler* plugin for Eclipse.

In Table 2 the total size of the generated code is expressed in kilobytes of Java code (*.java* **(KB)**) and kilobytes of executable file (*.class* **(KB)**) for the *UserAccess* and *Mondex* examples, respectively indicated in the rows labeled by *UA* and *M*.

Table 3 shows kilobytes of on-card files (*.cap+.exp* **(KB)**). The last row indicates the percentage of growth in card memory needs of the different compiled versions of the specification with respect to the verification free code.

A conclusion to be drawn from the experiments is that, depending on the resources available, different compilation options are recommended. If memory is the more critical resource in an application, the *-wrapper* (**JCMLc 1**) can be chosen, although **JCMLc 2** and **JCMLc 3** present similar behaviours, concerning the size occupied in the card. Otherwise, if processing time is the critical bottleneck, the *-inLineAll* (**JCMLc4**) approach is the more indicated, but it almost doubles memory needs. The best compromise between memory and execution times is obtained with option **-inLinePrePost**, or **JCMLc 3**. Because verification of pre- and postconditions is not replicated in the same way as it happens with the invariant, the inlining of these verification methods generates no extra code with respect to the non-inlined versions.

The data shown in these tables is depicted in Figs. 17 and 18.

Fig. 17(a) shows the size (in kilobytes) of the code generated for the *UserAccess* example. Notice that the times corresponding to the JCML programs are much nearer to the execution time for the original program (without annotations) than to those of JML. Fig. 17(b) shows the same data as in Fig. 17(a), but without the data for JML, in order to better show the relative execution times for the original files, as well as for each version of the JCML-generated code. Notice that the execution times for our JCMLc 4 implementation are quite similar to those of the original (no annotations) program. From these graphics, we can conclude that JCML is a viable solution for the specification of Java Card programs.

Fig. 18(a) shows the size (in kilobytes) of the code generated for the *UserAccess* example. From these graphics, it can be seen that the size of the generated files for JCML is both similar to the original Java Card files and much smaller than the code generated by JML. Fig. 18(b) shows the same data as in Fig. 18(a) where the data for JML has been omitted, in order to show the sizes of the original files, when compared to JCML.

The code generated by JCMLc (i) is, in all cases, much smaller than the one generated by JMLc and (ii) depending on the optimization level, may have a size which is similar to the original annotated program when we consider the on-card files.

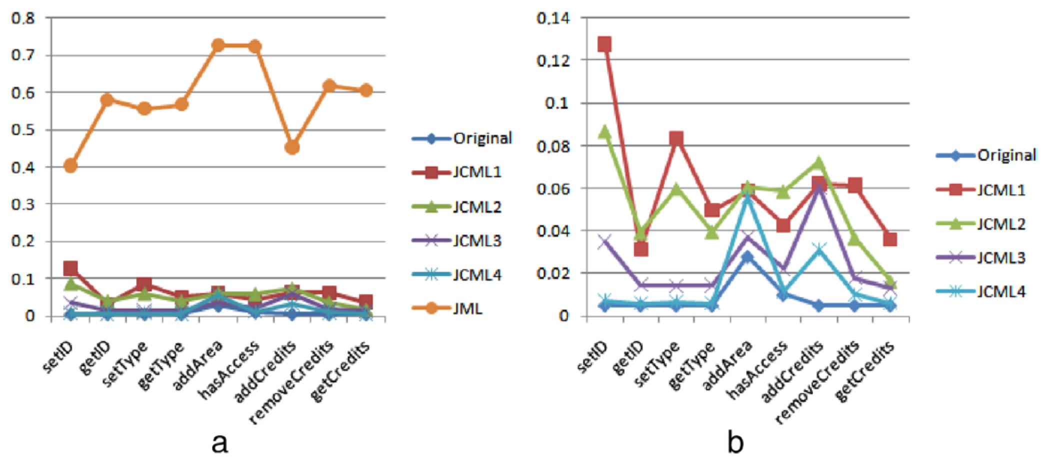


Fig. 17. Execution times (in KB).

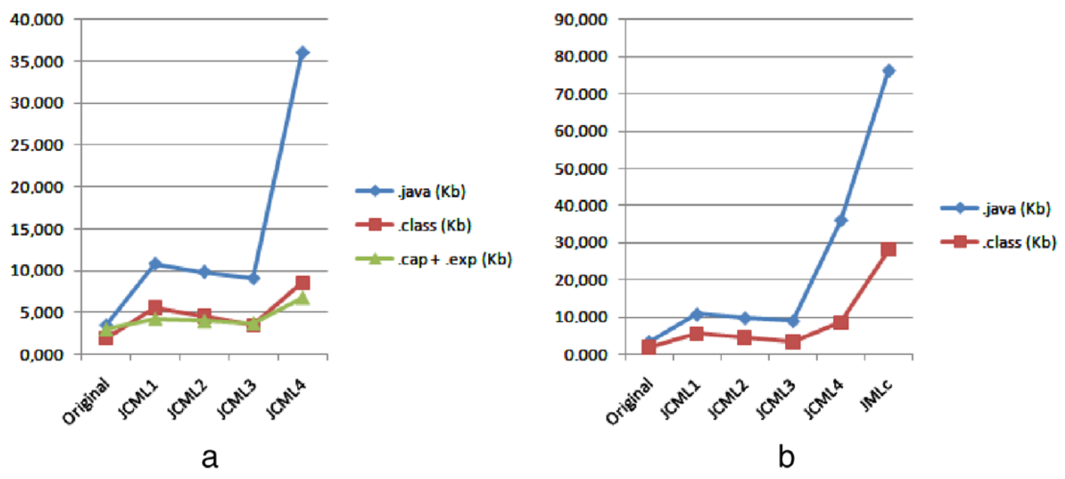


Fig. 18. Generated code size (in KB).

We have also experimented with five classes presented in [33,21]:

- *Animal* — Class for representing animals, with constructor and methods for consulting the gender of the animal and for setting its age.
- *Person* — Extension of the Animal class, with the same methods.
- *Patient* — Extension of the Person class, with the same methods.
- *StackAsArray* — Class for representing stacks, with constructor and push, pop, top, isEmpty and length methods.
- *IntMathOps* — Class with a method for returning square roots.

Those programs have been originally specified with JML and their verification codes have been produced by JMLc. Because such programs are not Java Card compliant, we are not able to produce *.cap* and *.exp* files. The experimental results presented in Table 4 show that the codes produced by JCMLc for those programs and their original specifications are also significantly smaller than the ones produced by JMLc, for all optimization levels. In Table 2 the size of the *.class* files are given in kilobytes. Notice that the size of files decreases as more methods are inlined in the produced code. Different from the *UserAccess* example, the size of code also reduces from the **JCMLc3** to the **JCMLc4** version, since we do not have invariant specification in those programs (except in the Patient class, but the invariant is very simple).

One important note to make here, is that some of this verification code (at least the one corresponding to the preconditions) would be included by the developer anyway, as this is the usual Java Card robust programming style.

The execution times, together with the size of the on-card files for the programs we have experimented with, show that the use of JCML is both possible for Java Card and not too expensive. The code generated by our compiler is consistently faster and smaller than the code generated by JMLc. The facts stated above allow us to conclude that one can afford the use of a behavioural specification language on Java Card.

Table 4
Animal, Person, Patient, InMathOps and Stack class sizes (in KB).

Class	Original	JCMLc 1	JCMLc 2	JCMLc 3	JCMLc 4	JMLc
Animal	0.757	2.40	1.98	1.30	1.30	15.50
Person	0.742	2.25	1.96	1.29	1.29	18.00
Patient	0.589	2.06	1.96	1.49	1.48	12.70
InMathOps	0.395	1.06	0.99	0.65	0.65	9.38
StackAsArray	0.751	2.60	2.15	1.27	1.27	19.80

7. Related work

The problem of building reliable Java Card applications has many features in common with its counterpart for the Java programming language. However, the memory and processing limitations of our setting lead to some important differences among these problems. A number of challenges (and proposed solutions) for the specification and verification for object-oriented programs are addressed in [24] . Some of these challenges are shared with our problem (like those dealing with the need for formal tools to study the effect of methods on static variables; the need for tools to deal with finalizers or the need for assistance in specifying libraries of classes). Other challenges in [24], mostly dealing with partial specifications and with complex data, are not related to our language.

In a recent paper [8], Chalin and Rioux report a change in the evaluation of expressions present in JML assertions. The new semantics simplifies the interpretation undefined expressions, by using an approach similar to the Java language. These kinds of semantics were adopted by our tool since their first implementation, since the generated code for expressions is much simpler than that for the original JML proposal [10].

There exist some systems that address the formal treatment of Java Card. Two of these systems are the most popular among developers: Krakatoa [27] and the KeY System [6,5]. Krakatoa proves Java/Java Card programs annotated with JML specifications by using the ***Why*** [14] and ***Coq*** [37] tools. ***Why*** is a proof obligation generator and ***Coq*** is a proof assistant. Krakatoa translates Java/Java Card code into the ***Why*** input language (an ML-like language), which generates proof obligations to be interactively proved by means of the ***Coq*** proof assistant. The KeY system is intended to integrate the design, implementation and formal specification and verification of object-oriented languages. The KeY system is based on a theorem prover for the first-order Dynamic Logic for Java and can verify Java Card programs thoroughly. Both Krakatoa and KeY perform static verification only, as it is the case of other related works [25,31,39].

Efforts towards runtime verification of Java (and Java ME) can be found in [33]. That work proposes the use of AspectJ to implement a JML compiler that takes specifications and generates bytecode compliant with both Java and Java ME virtual machines. Regarding their language constructs, Java ME is a richer language than Java Card and the architectures for which Java ME is targeted are less constrained than those in which Java Card applets run.

8. Conclusions and future work

This paper presented JCML – a language for the specification of Java Card programs – and its associated compiler. JCML annotates Java Card programs to produce runtime verification code which can be performed on devices with severe memory and processing restrictions. JCML includes all JML constructs which can be translated into Java Card compliant code. A case study was used, and the obtained results show that the proposed approach is effective. For instance, in our example, JCML generated an executable code which is approximately 75% smaller than the one generated by JMLc, even when execution speed is the primary concern (all verification methods inlined).

The code generated by our compiler is smaller and faster than equivalent code generated by the original JML compiler. This is due to the following facts:

- JCMLc is devised to be optimized: For instance, no tests or calls are generated for empty conditions. The original JML compiler generates code for all conditions, independently of the original JML specification.
- We use static analysis to define the upper and lower bound of variables in quantifiers.
- In the original JML compiler, assertion undefinedness is treated in such a way that assertion runtime checking considers the context and the kind of event that led to the exception to conservatively preserve JML semantics avoiding false positives as much as possible. This complex treatment given by JMLc is however too heavy for a smart card environment. Our choice was then to assume that any uncheckable specification is false.

The version of the compiler presented here implements all those primitives of JCML level 0 that are meaningful in the context of Java Card, as well as some constructs of other levels. Our compiler allows the user to choose the amount of method inlining for each unit. This feature can be used to choose the most adequate setting for each application.

As a future work, we plan to deal with exceptional behaviour so that the application that runs on-card is able to gracefully recover from faults.

In another line of work, studies concerning the use of aspects in the implementation of the compiler, as done in [33], will also be carried out. We plan to compare results with our current approach and to identify possible common features and improvements.

Acknowledgements

The authors would like to thank the SCP and SBMF reviewers, as well as those SBMF2008 attendants who provided valuable feedback on this work. Our thanks, in particular, to Prof. Jim Woodcock, for his important observations about our treatment of undefined expressions.

References

[1] J.-R. Abrial, The B Book: Assigning Programs to Meanings, Cambridge University Press, 1996.
[2] Cyrille Artho, Finding faults in multi-threaded programs. Technical report, Master's thesis, 2001.
[3] Cyrille Artho, Armin Biere, Combined static and dynamic analysis, Electronic Notes in Theoretical Computer Science 131 (2005) 3–14.
[4] Emine G. Aydal, Richard F. Paige, Jim Woodcock, Evaluation of OCL for large-scale modelling: a different view of the Mondex purse, in: ECEASST, 2008, p. 9.
[5] Bernhard Beckert, Martin Giese, Reiner Hähnle, Vladimir Klebanov, Philipp Rümmer, Steffen Schlager, Peter H. Schmitt, The KeY System 1.0 (Deduction Component), in: CADE, 2007, pp. 379–384.
[6] Bernhard Beckert, Reiner Hähnle, Peter H. Schmitt (Eds.), Verification of Object-Oriented Software: The KeY Approach, in: LNCS, vol. 4334, Springer-Verlag, 2007.
[7] A. Bhorkar, A run-time assertion checker for Java using JML, Technical report 00-08, Department of Computer Science, Iowa State University, May 2000.
[8] Patrice Chalin, Frédéric Rioux, Jml runtime assertion checking: improved error reporting and efficiency using strong validity, in: Jorge Cuéllar, T.S.E. Maibaum, Kaisa Sere (Eds.), FM, in: Lecture Notes in Computer Science, vol. 5014, Springer, 2008, pp. 246–261.
[9] Zhiqun Chen, Java Card Technology for Smart Cards: Architecture and Programmer's Guide, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
[10] Y. Cheon, A runtime assertion checker for the Java Modeling Language, PhD thesis, Department of Computer Science, Iowa State University, April 2003.
[11] Umberto S. Costa, Anamaria M. Moreira, Martin A. Musicante, Plácido A. Souza Neto, Specification and runtime verification of Java Card Programs, in: Brazilian Symposium on Formal Methods (SBMF), 2008.
[12] Umberto Souza da Costa, Anamaria Martins Moreira, Martin A. Musicante, Plácido A. Souza Neto, Specification and runtime verification of Java Card programs, Electronic Notes in Theoretical Computer Science 240 (2009) 61–78.
[13] Michael D. Ernst, Static and dynamic analysis: synergy and duality, in: Cormac Flanagan, Andreas Zeller (Eds.), PASTE, ACM, 2004, p. 35.
[14] J.-C. Filliâtre, Why: a multi-language multi-prover verification condition generation, Technical report, Université Paris-Sud, France, LRI - CNRS UMR 8623, March 2003.
[15] Leo Freitas, Jim Woodcock, Mechanising Mondex with Z/Eves, Formal Aspects of Computing 20 (1) (2008) 117–139.
[16] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns, Addison-Wesley Professional, 1995.
[17] Chris George, Anne Elisabeth Haxthausen, Specification, proof, and model checking of the Mondex electronic purse using RAISE, Formal Aspects of Computing 20 (1) (2008) 101–116.
[18] Cliff Jones, Peter O'Hearn, Jim Woodcock, Verified software: a grand challenge, IEEE Computer Magazine, 2006. http://www.computer.org/computer/author.htm.
[19] Cliff B. Jones, Jim Woodcock, Special edition on the Mondex case study, Formal Aspects of Computing 20 (1) (2008).
[20] T.M. Jurgensen, S.B. Guthery, Smart Cards: A Developer's Toolkit, Prentice Hall PTR, 2002.
[21] G.T. Leavens, Y. Cheon, Design by contract with JML. Draft. Available from http://www.jmlspecs.org, 2009.
[22] G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, JML Reference Manual, May 2008.
[23] G.T. Leavens, Tutorial on JML, the Java Modeling Language, in: ASE, 2007, p. 573.
[24] G.T. Leavens, K.R.M. Leino, Peter Müller, Specification and verification challenges for sequential object-oriented programs, Formal Aspects of Computing 19 (2) (2007) 159–189.
[25] K.R.M. Leino, G. Nelson, J.B. Saxe, ESC/Java user's manual. Technical note, Compaq Systems Research Center, October 2000.
[26] K.R.M. Leino, Peter Müller, Object invariants in dynamic contexts, in: Martin Odersky (Ed.), ECOOP, in: Lecture Notes in Computer Science, vol. 3086, Springer, 2004, pp. 491–516.
[27] C. Marche, C. Paulin Mohring, X. Urbain, The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML, Journal of Logic and Algebraic Programming 58 (1–2) (2004) 89–106.
[28] Bertrand Meyer, Applying "design by contract", IEEE Computer 25 (10) (1992) 40–51.
[29] Bertrand Meyer, Object-Oriented Software Construction, Prentice Hall PTR, 2000.
[30] SUN Microsystems. API specification — Java Card platform application programming interface specification, Version 2.2.1. Sun Microsystems, 2005.
[31] Jeremy W. Nimmer, Michael D. Ernst, Static verification of dynamically detected program invariants: integrating Daikon and Esc/java, 2001.
[32] E. Ortiz, An introduction to Java Card technology, Technical report, Sun Microsystems, 2005.
[33] H. Rebêlo, S. Soares, M. Cornélio, R. Lima, L. Ferreira, Implementing Java modeling language contracts with AspectJ, in: Proceedings of the 23rd Annual ACM Symposium on Applied Computing, Fortaleza-Brazil, 2008, pp. 228–233.
[34] Peter H. Schmitt, Isabel Tonin, Verifying the Mondex case study, in: Mike Hinchey, Tiziana Margaria (Eds.), Proc. 5.IEEE Int.Conf. on Software Engineering and Formal Methods (SEFM), IEEE Press, 2007, pp. 47–56.
[35] Yannis Smaragdakis, Christoph Csallner, Combining static and dynamic reasoning for bug detection, in: Yuri Gurevich, Bertrand Meyer (Eds.), TAP, in: Lecture Notes in Computer Science, vol. 4454, Springer, 2007, pp. 1–16.
[36] Plácido A. Souza Neto, JCML — Java Card Modeling Language: definition and implementation. Master's thesis, Programa de Pós-Graduação em Sistemas e Computação, Universidade Federal do Rio Grande do Norte, 2007. (In portuguese).
[37] The Coq Development Team. The Coq proof assistant: reference manual: Version 8.1. INRIA, 2007. Available at http://coq.inria.fr/doc-eng.html.
[38] Isabel Tonin, Verifying the Mondex case study. The key approach. Techischer Bericht 2007-4, Fakultät für Informatik, Universität Karlsruhe, 2007.
[39] J. Van den Berg, B. Jacobs, The LOOP compiler for Java and JML, in: Tiziana Margaria, Wang Yi (Eds.), TACAS, in: Lecture Notes in Computer Science, vol. 2031, Springer, 2001, pp. 299–312.
[40] Jim Woodcock, Jim Davies, Using Z: Specification, Refinement, and Proof, in: Series in Computer Science, Prentice Hall International, Upper Saddle River, NJ, USA, 1996.
[41] Jim Woodcock, Leo Freitas, Z/eves and the Mondex electronic purse, in: Kamel Barkaoui, Ana Cavalcanti, Antonio Cerone (Eds.), ICTAC, in: Lecture Notes in Computer Science, vol. 4281, Springer, 2006, pp. 15–34.